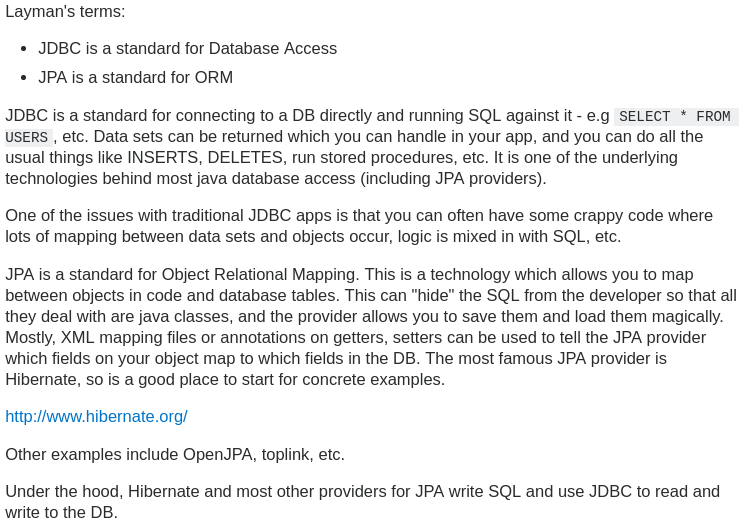
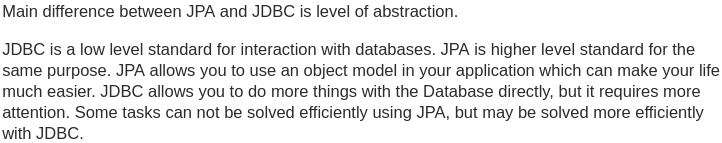
**The difference between JPA and JDBC**

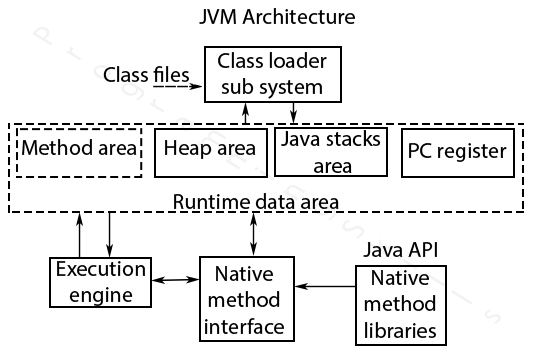




→ JDBC is already implemented by Java. JPA is a standard/specification which different providers implement.

**Differences between JDK, JRE and JVM**

JVM:

**1. Class Loader Subsystem**

Java's dynamic class loading functionality is handled by the class loader subsystem. It loads, links. and initializes the class file when it refers to a class for the first time at runtime, not compile time.

**1.1 Loading**

Classes will be loaded by this component. Boot Strap class Loader, Extension class Loader, and Application class Loader are the three class loader which will help in achieving it.

* **Boot Strap ClassLoader** – Responsible for loading classes from the bootstrap classpath, nothing but rt.jar. Highest priority will be given to this loader.
* **Extension ClassLoader** – Responsible for loading classes which are inside ext folder (jre\lib).
* **Application ClassLoader** –Responsible for loading Application Level Classpath, path mentioned Environment Variable etc.

The above Class Loaders will follow Delegation Hierarchy Algorithm while loading the class files.

**1.2 Linking**

* **Verify** – Bytecode verifier will verify whether the generated bytecode is proper or not if verification fails we will get the verification error.
* **Prepare** – For all static variables memory will be allocated and assigned with default values.
* **Resolve** – All symbolic memory references are replaced with the original references from Method Area.

**1.3 Initialization**

This is the final phase of Class Loading, here all static variables will be assigned with the original values, and the static block will be executed.

**2. Runtime Data Area**

The Runtime Data Area is divided into 5 major components:

* **Method Area** – All the class level data will be stored here, including static variables. There is only one method area per JVM, and it is a shared resource.
* **Heap Area** – All the Objects and their corresponding instance variables and arrays will be stored here. There is also one Heap Area per JVM. Since the Method and Heap areas share memory for multiple threads, the data stored is not thread safe.
* **Stack Area** – For every thread, a separate runtime stack will be created. For every method call, one entry will be made in the stack memory which is called as Stack Frame. All local variables will be created in the stack memory. The stack area is thread safe since it is not a shared resource. The Stack Frame is divided into three sub-entities:
  + Local Variable Array – Related to the method how many local variables are involved and the corresponding values will be stored here.
  + Operand stack – If any intermediate operation is required to perform, operand stack acts as runtime workspace to perform the operation.
  + Frame data – All symbols corresponding to the method is stored here. In the case of any exception, the catch block information will be maintained in the frame data.
* **PC Registers** – Each thread will have separate PC Registers, to hold the address of current executing instruction once the instruction is executed the PC register will be updated with the next instruction.
* **Native Method stacks** – Native Method Stack holds native method information. For every thread, a separate native method stack will be created.

**3. Execution Engine**

The bytecode which is assigned to the Runtime Data Area will be executed by the Execution Engine. The Execution Engine reads the bytecode and executes it piece by piece.

* **Interpreter** – The interpreter interprets the bytecode faster, but executes slowly. The disadvantage of the interpreter is that when one method is called multiple times, every time a new interpretation is required.
* **JIT Compiler** – The JIT Compiler neutralizes the disadvantage of the interpreter. The Execution Engine will be using the help of the interpreter in converting byte code, but when it finds repeated code it uses the JIT compiler, which compiles the entire bytecode and changes it to native code. This native code will be used directly for repeated method calls, which improve the performance of the system.
  + Intermediate Code generator – Produces intermediate code
  + Code Optimizer – Responsible for optimizing the intermediate code generated above
  + Target Code Generator – Responsible for Generating Machine Code or Native Code
  + Profiler – A special component, responsible for finding hotspots, i.e. whether the method is called multiple times or not.
* **Garbage Collector** - Collects and removes unreferenced objects. Garbage Collection can be triggered by calling "**System.gc()**", but the execution is not guaranteed. Garbage collection of the JVM collects the objects that are created.

**JIT Compiler**

A JIT compiler runs **after** the program has started and compiles the code (usually bytecode or some kind of VM instructions) on the fly (or just-in-time, as it's called) into a form that's usually faster, typically the host CPU's native instruction set. A JIT has access to dynamic runtime information whereas a standard compiler doesn't and can make better optimizations like inlining functions that are used frequently.

This is in contrast to a traditional compiler that compiles **all** the code to machine language **before** the program is first run.

To paraphrase, conventional compilers build the whole program as an EXE file BEFORE the first time you run it. For newer style programs, an assembly is generated with pseudocode (p-code). Only AFTER you execute the program on the OS (e.g., by double-clicking on its icon) will the (JIT) compiler kick in and generate machine code (m-code) that the Intel-based processor or whatever will understand.

Typical scenario: The source code is completely converted into machine code

JIT scenario: The source code will be converted into assembly language like structure [for ex IL (intermediate language) for C#, ByteCode for java].

The intermediate code is converted into machine language only when the application needs that is required codes are only converted to machine code.

JIT vs Non-JIT comparison:

* In JIT not all the code is converted into machine code first a part of the code that is necessary will be converted into machine code then if a method or functionality called is not in machine then that will be turned into machine code... it reduces burden on the CPU.
* As the machine code will be generated on run time....the JIT compiler will produce machine code that is optimised for running machine's CPU architecture.

JIT Examples: In Java JIT is in JVM (Java Virtual Machine) In C# it is in CLR (Common Language Runtime) In Android it is in DVM (Dalvik Virtual Machine), or ART (Android RunTime) in newer versions.

**A compiler would create machine language**

No. A compiler is simply a program which takes as its input a program written in language **A** and produces as its output a semantically equivalent program in language **B**. Language **B** can be anything, it can be but doesn't have to be machine language.

**which runs on the physical hardware directly?**

Not necessarily. It could be run in an interpreter or in a VM. It could be further compiled to a different language.

I think you should drop the notion of "compiler versus interpreter" entirely, because it's a false dichotomy.

* A **compiler** is a transformer: It transforms a computer program written in a source language and outputs an equivalent in a target language. Usually, the source language is higher-level that the target language - and if it's the other way around, we often call that kind of transformer a decompiler.
* An **interpreter** is an execution engine. It executes a computer program written in one language, according to the specification of that language. We mostly use the term for software (but in a way, a classical CPU can be viewed as a hardware-based "interpreter" for its machine code).

Why does Java typically interpret instead of compile? The main advantage of compilation is that you end up with raw machine language code that can be efficiently executed on your machine. However, it can only be executed on one type of machine architecture (Intel Pentium, PowerPC). A primary advantage of a compiling to an intermediate language like Java bytecode and then interpreting is that you can achieve platform independence: you can interpret the same .class file on differently types of machine architectures. However, interpreting the bytecode is typically slower than executing pre-compiled machine language code. A second advantage of using the Java bytecode is that it acts as a buffer between your computer and the program. This enables you to download an untrusted program from the Internet and execute it on your machine with some assurances. Since you are running the Java interpreter (and not raw machine language code), you are protected by a layer of security which guards against malicious programs. It is the combination of Java and the Java bytecode that yield a platform-independent and secure environment, while still embracing a full set of modern programming abstractions.

The Java bytecode and the java interpreter are not inherently specific to the Java programming language. For example, you can use Jython to compile from the Python programming language into Java bytecode, and then use java to interpret it. There are similar ML, Lisp, and Fortran compilers that compile into JAva bytecode. You could also use the Unix program gcj to compile directly from a .java source file into a machine executable file a.out, which can be run natively on any Sparc microprocessors. Additionally, you can design hardware whose machine language is the Java bytecode. Sun Microsystems has done exactly this, making the Java Virtual Machine not so virtual.

Why not use a real machine language instead of the Java bytecode? The Java bytecode is much simpler than a typical high-level programming language. It is much easier to write a Java bytecode interpreter for a new type of computer than it is to write a full Java compiler. The abstraction means that it is much easier to reason about security and performance.

**Why auto-import only java.lang package?**

A good reason not to autoimport too much is to avoid namespace clashes. If everything in java.util was imported automatically and then you wanted to refer to a different class named 'Map', for example, you would have to refer to it by its fully-qualified name.

Provides classes that are fundamental to the design of the Java programming language. The most important classes are Object, which is the root of the class hierarchy, and Class, instances of which represent classes at run time. Wrapper classes, Math, Number, ProcessBuilder, Runtime …

**How does Java import work?**

In Java, import is simply used by the compiler to let you name your classes by their unqualified name, let's say String instead of java.lang.String. You don't really need to import java.lang.\* because the compiler does it by default. However this mechanism is just to save you some typing. Types in Java are fully qualified class names, so a String is really a java.lang.String object when the code is run. Packages are intended to prevent name clashes and allow two classes to have the same simple name, instead of relying on the old C convention of prefixing types like this. java\_lang\_String. This is called **namespacing**. Hatırla, aynı paketde ise import’a gerek yok çünkü o classı görüyor **==** aynı namespace

BTW, in Java there's the **static import** construct, which allows to further save typing if you use lots of constants from a certain class. In a compilation unit (a .java file) which declares

**import static** java.lang.Math.\*;

**→** Classın içindeki bütün constantları ve methodlara direk erişebiliyorsun.

you can use the constant PI in your code, instead of referencing it through Math.PI, and the method cos() instead of Math.cos(). So for example you can write

double r = cos(PI \* theta);

Java's import statement is pure syntactical sugar. import is only evaluated at **compile time** to indicate to the compiler where to find the names in the code.

You may live without any import statement when you always specify the full qualified name of classes. Like this line needs no import statement at all:

javax.swing.JButton but = new javax.swing.JButton();

The import statement will make your code more readable like this:

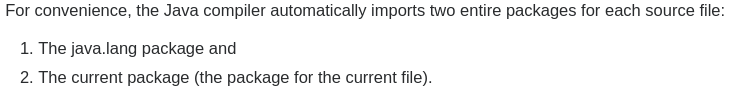
import javax.swing.\*;

JButton but = new Jbutton();

* java.lang paketini auto-import edildiği için kısa versiyonunu yazıyoruz. Ancak bu şekilde de yazılabiliyor, aynısı başka paketler için de geçerli:



* Ancak import ettiğimiz için, Math veya Integer veya Double yazdığımızda, ne olduğunu biliyor, bize “bu nedir bilmiyorum” diye bir hata vermiyor. Aynısı bütün paketler için geçerli. **Namespace** denilen olay bu.



“Library Set” içinde neler var?

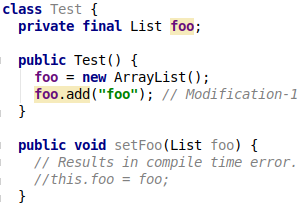
“Development tools” hangileri?

Tomcat javada yazılmış ve kendi VM i var ama tomcati hangi vm çalıştırıyor?

VM çalışınca instance mı oluşuyor? Bu durumda 2 VM instanceı mı oluyor 1 I tomcati çalıştıran, diğeri tomcatin içindeki servletları çalıştıran?

**Final Keyword**

This is one of the favorite interview questions. With this question, the interviewer tries to find out how well you understand the behavior of objects with respect to constructors, methods, class variables (static variables) and instance variables.



In the above case, we have defined a constructor for 'Test' and gave it a 'setFoo' method.

**About constructor**: Constructor can be invoked only one time per object creation by using the new keyword. You cannot invoke constructor multiple times, because constructor are not designed to do so.

**About method**: A method can be invoked as many times as you want (Even never) and the compiler knows it.

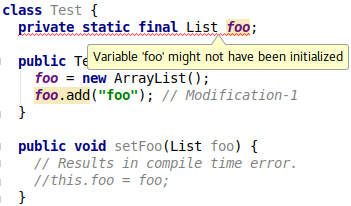
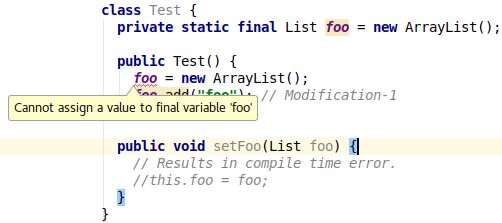
**Scenario 1**

foo is an **instance** variable. When we create Test class object then the instance variable foo, will be copied inside the object of Test class. If we assign foo inside the constructor, then the compiler knows that the constructor will be invoked only once, so there is no problem assigning it inside the constructor.

If we assign foo inside a method, the compiler knows that a method can be called multiple times, which means the value will have to be changed multiple times, which is not allowed for a final variable. So the compiler decides constructor is good choice! **You can assign a value to a final variable only one time**.

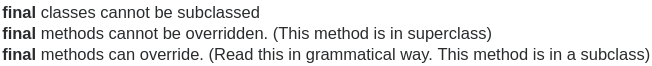
**Scenario 2**

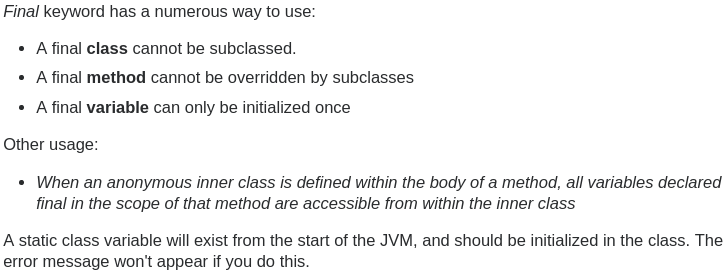
foo is now a **static** variable. When we create an instance of Test class, foo will not be copied to the object because foo is static. Now foo is not an independent property of each object. This is a property of Test class. But foo can be seen by multiple objects and if every object which is created by using the new keyword which will ultimately invoke the Test constructor which changes the value at the time of multiple object creation (Remember static foo is not copied in every object, but is shared between multiple objects.)

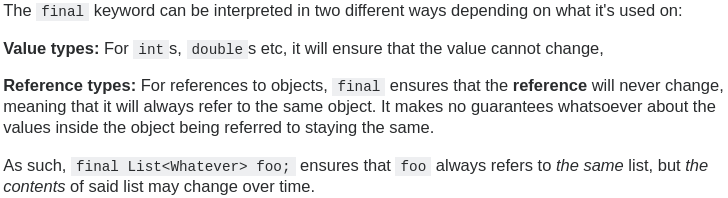


**Scenario 3**

Above Modification-2 is from your question. In the above case, you are not changing the first referenced object, but you are adding content inside foo which is allowed. Compiler complains if you try to assign a new ArrayList() to the foo reference variable. Rule If you have initialized a final variable, then you cannot change it to refer to a different object. (In this case ArrayList)





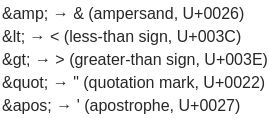


**Hatırlatma**:

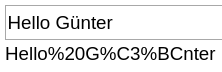
* **URL escaping** ve **HTML escaping** farklı şeyler.

Mantıkları aynı. Programlama dillerinde de farklı karakterler escape ediliyor.

Örnek: Java’da **char escaping** var. ekrana **\t** yazdırmak istiyoruz ancak **print(“\t”)** yazarsak programlama dili onu bizim isteiğimiz gibi anlamayacak, o yüzden **print(“\\t”)** yazmak gerek. **\”** , **\’** …



**HTML escaping**: ekrana <h3> yazdırmak istiyoruz ancak html parser onu “yanlış” anlayacak. O yüzden escape etmemiz gerek.

**URL escaping / encoding (Percent Encoding)**: ASCII olmayan her şeyi ve boşlukları vs encode etmemiz lazım. UTF-8 hex karşılığı ile sanırım.

URLs can only be sent over the Internet using the ASCII character-set.

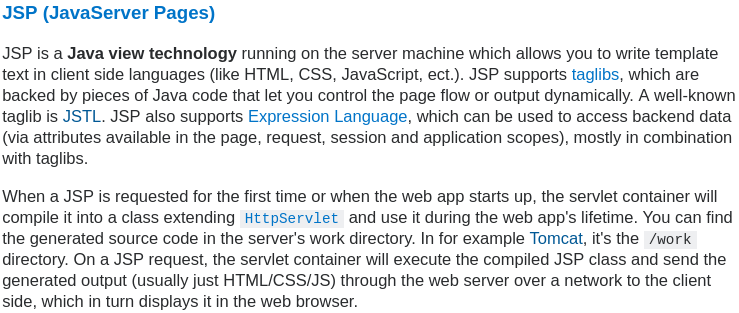
Since URLs often contain characters outside the ASCII set, the URL has to be converted into a valid ASCII format.

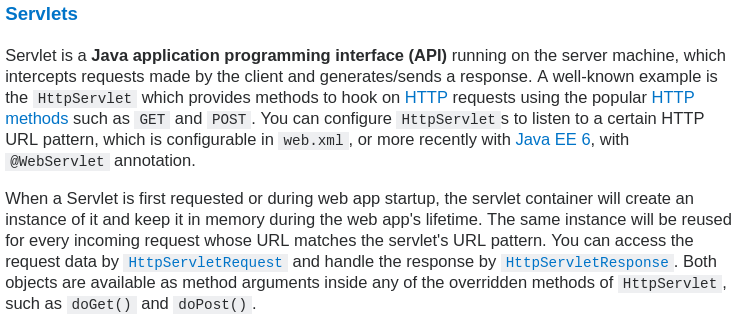
URL encoding replaces unsafe ASCII characters with a "%" followed by two hexadecimal digits.

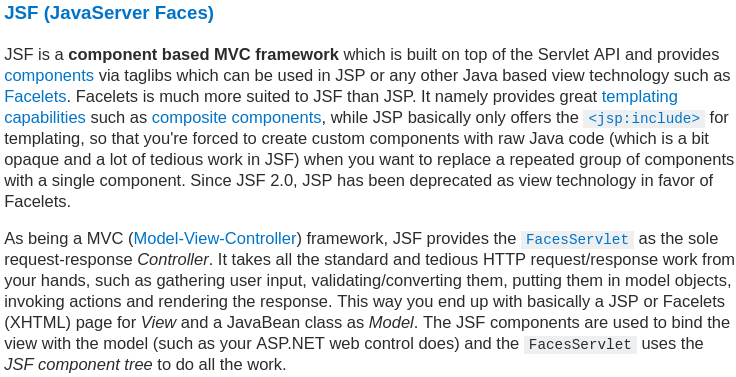
URLs cannot contain spaces. URL encoding normally replaces a space with a plus (+) sign or with %20.

**→ ASCII’**de **ü** olmadığı için, ü’yü ascii tabelasındaki harflerle nasıl gösterebiliriz? **Ü**’nün **unicode** **karşılığını** **UTF-8 formatı** ile, yani hex değerleri ile gösteririz. % ve bütün hex değerleri ASCII var.

**What is the difference between JSF, Servlet and JSP?**



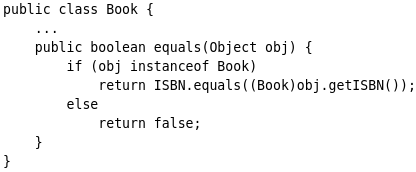




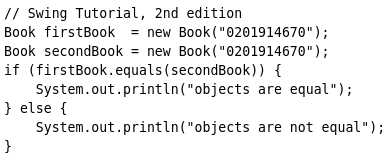
**The equals() Method**

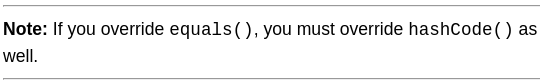
The equals() method compares two objects for equality and returns true if they are equal. **The equals() method provided in the Object class uses the identity operator (==) to determine whether two objects are equal**. For primitive data types, this gives the correct result. For objects, however, it does not.

The equals() method provided by Object **tests whether the object references are equal**—that is, if the objects compared are the exact same object. To test whether two objects are equal in the sense of equivalency (containing the same information), you **must override the equals()** method. Here is an example of a Book class that overrides equals():

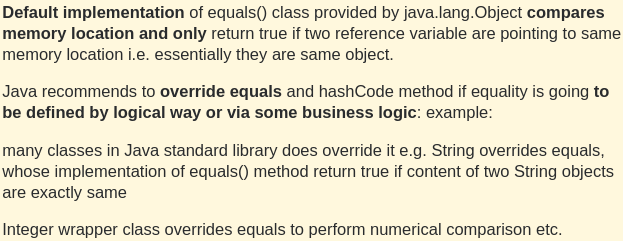
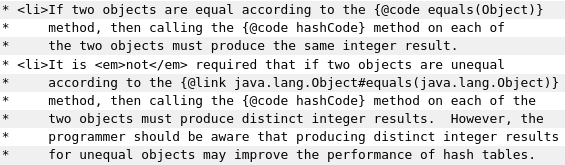


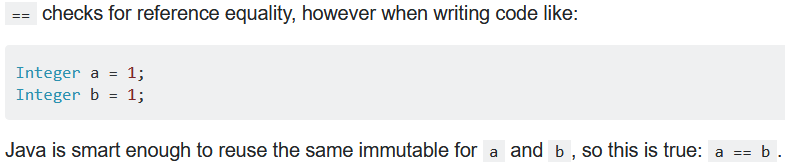
Consider this code that tests two instances of the Book class for equality:

This program displays objects are equal even though firstBook and secondBook reference two distinct objects. They are considered equal because the objects compared contain the same ISBN number. You should always override the equals() method if the identity operator is not appropriate for your class.



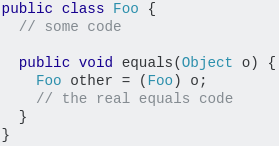
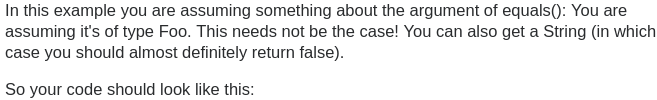
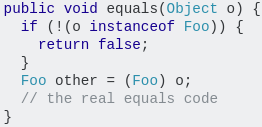


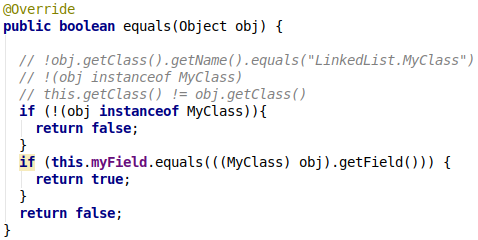


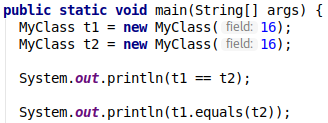


Graphical user interface, text, application

Description automatically generated





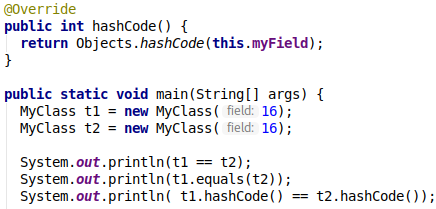


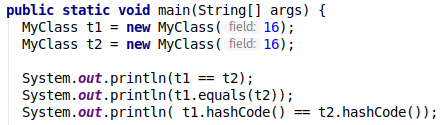


**The hashCode() Method**

The value returned by hashCode() is the object's hash code, **which is the object's memory address in hexadecimal**. (Object class implementation)

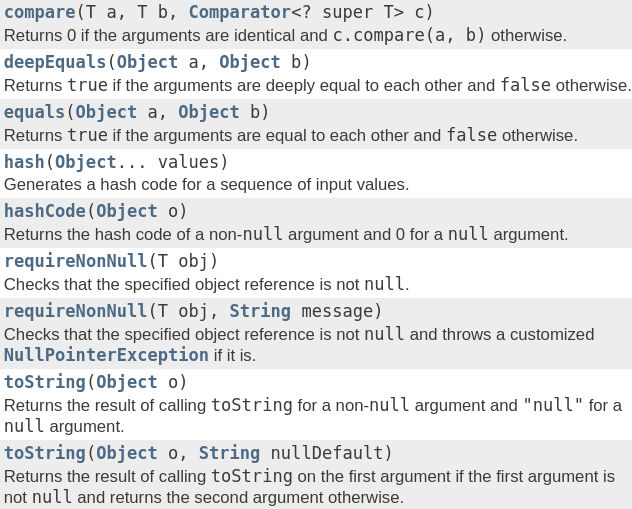
By definition, if two objects are equal, their hash code must also be equal. If you override the equals() method, you change the way two objects are equated and Object's implementation of hashCode() is no longer valid. Therefore, if you override the equals() method, you must also override the hashCode() method as well.







As we know, if two objects are equal then their hashCode must also be equal. In the first example we did not override the hashCode() method, we only provided an equals(). We see that .equals() and hashCode() resulted in different booleans. The second example shows how it should be.





This class consists of static utility methods for operating on objects. These utilities include null-safe or null-tolerant methods for computing the hash code of an object, returning a string for an object, and comparing two objects.

**Understanding How hashCode() Works**

Simply put, hashCode() returns an integer value, generated by a hashing algorithm.

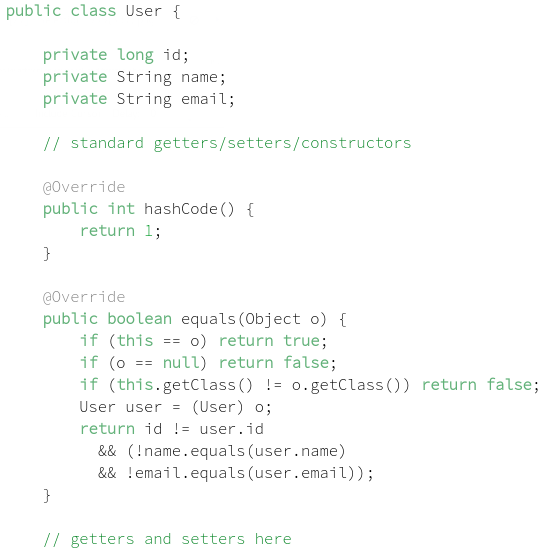
Objects that are equal (according to their equals()) must return the same hash code. **It’s not required for different objects to return different hash codes**.

The general contract of hashCode() states:

* Whenever it is invoked on the same object more than once during an execution of a Java application, hashCode() must consistently return the same value, provided no information used in equals comparisons on the object is modified. This value needs not remain consistent from one execution of an application to another execution of the same application
* If two objects are equal according to the equals(Object) method, then calling the hashCode() method on each of the two objects must produce the same value
* It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, developers should be aware that producing distinct integer results for unequal objects improves the performance of hash tables

**A Naive hashCode() Implementation**

It’s actually quite straightforward to have a naive hashCode() implementation that fully adheres to the above contract. To demonstrate this, we’re going to define a sample User class that overrides the method’s default implementation:



The User class provides custom implementations for both **equals()** and **hashCode()** that fully adhere to the respective contracts. Even more, there’s nothing illegitimate with having hashCode() returning any fixed value.

However, this implementation degrades the functionality of hash tables to basically zero, as every object would be stored in the same, single bucket.

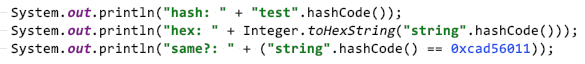
In this context, a hash table lookup is performed linearly and does not give us any real advantage – more on this in section 7.

While it’s essential to understand the roles that hashCode() and equals() methods play, we don’t have to implement them from scratch every time, as most IDEs can generate custom hashCode() and equals() implementations and since **Java 7**, we got an **Objects.hash()** utility method for comfortable hashing.

* **An object’s hashCode method must take the same fields into account as its equals method**.

Graphical user interface, text, application

Description automatically generated



Text

Description automatically generated with medium confidence



Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, Teams

Description automatically generated

**Handling Hash Colisions**

The intrinsic behavior of hash tables raises up a relevant aspect of these data structures: even with an efficient hashing algorithm, two or more objects might have the same hash code, even if they’re unequal. So, their hash codes would point to the same bucket, even though they would have different hash table keys.

This situation is commonly known as a hash collision, and various methodologies exist for handling it, with each one having their pros and cons. Java’s HashMap uses the separate chaining method for handling collisions:

**“When two or more objects point to the same bucket, they’re simply stored in a linked list. In such a case, the hash table is an array of linked lists, and each object with the same hash is appended to the linked list at the bucket index in the array.**

**In the worst case, several buckets would have a linked list bound to it, and the retrieval of an object in the list would be performed linearly.”**

Hash collision methodologies show in a nutshell why it’s so important to implement hashCode() efficiently.

Java 8 brought an interesting enhancement to HashMap implementation – if a bucket size goes beyond the certain threshold, the linked list gets replaced with a tree map. This allows achieving O(logn) look up instead of pessimistic O(n).

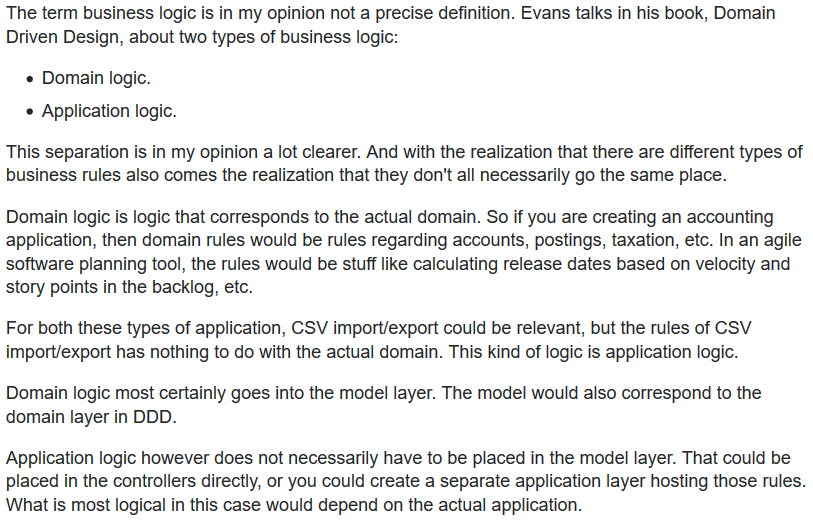
Text

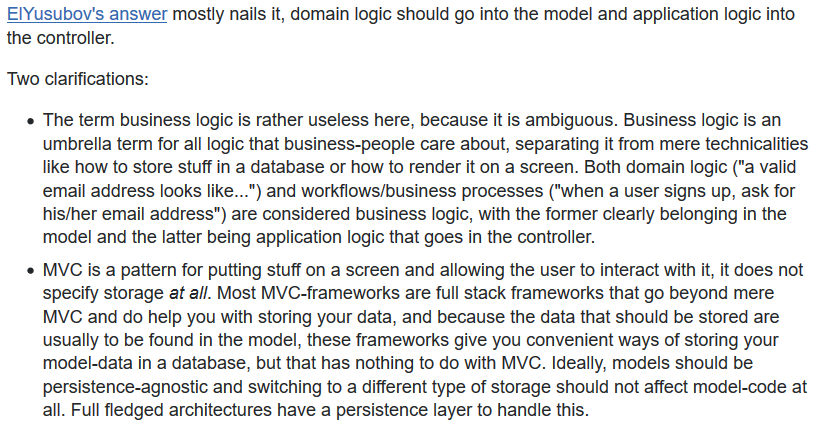
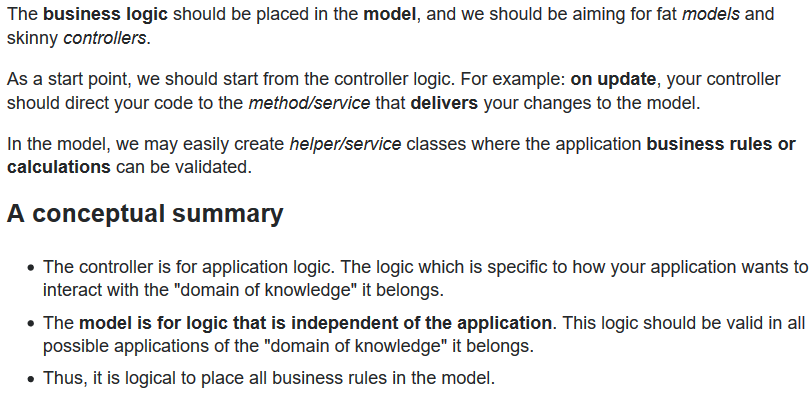
Description automatically generated

Text

Description automatically generated





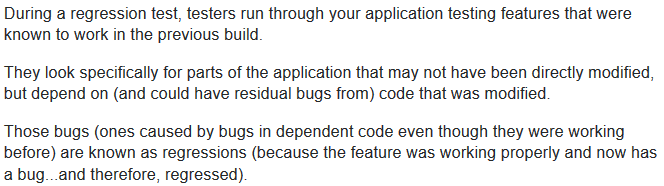


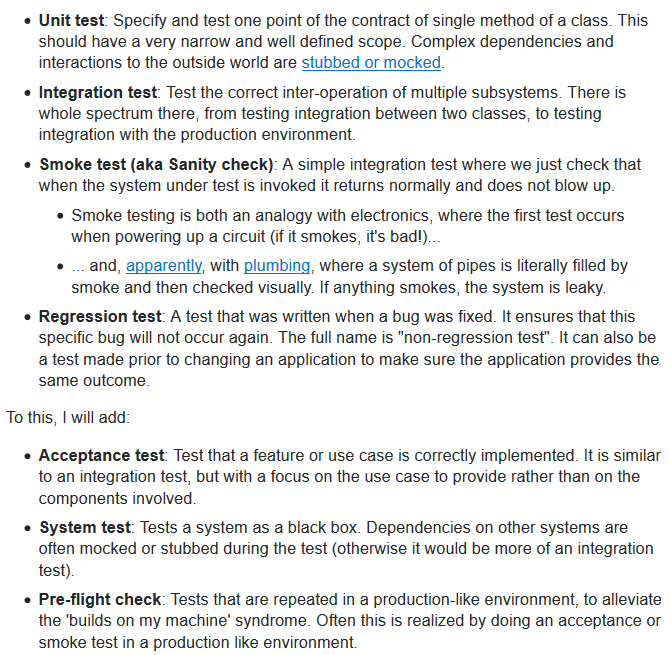
**Testing**

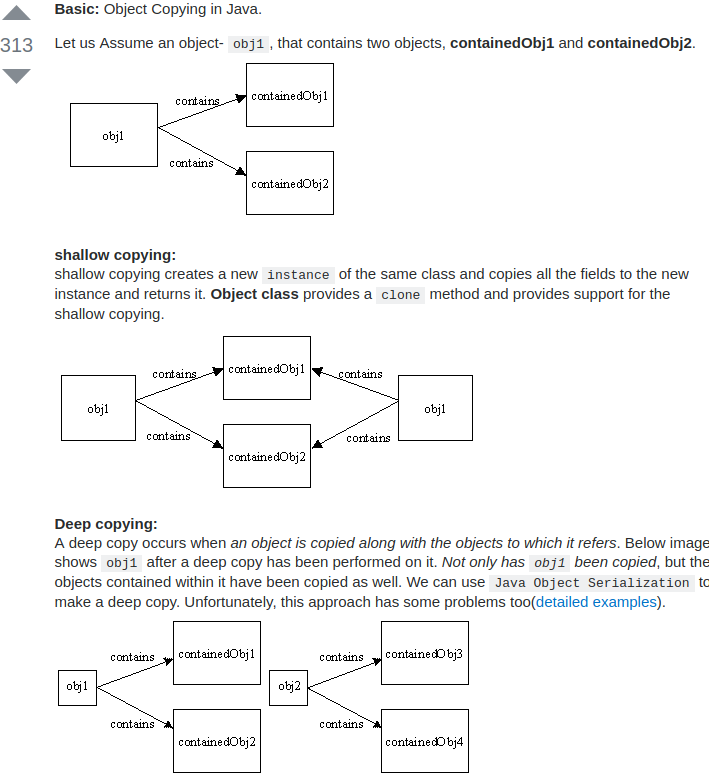
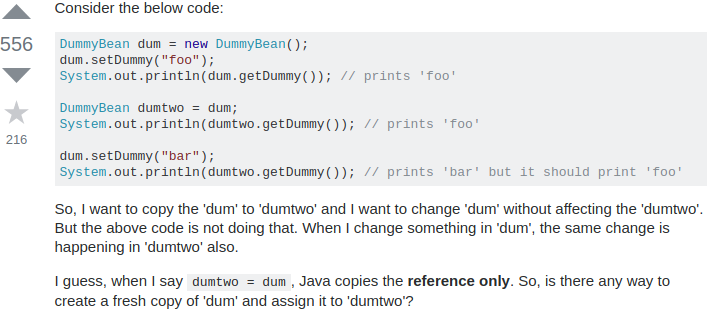
**Regression Test**

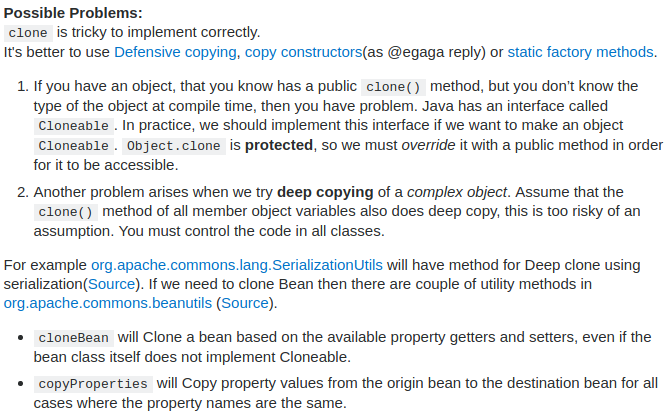
Regression test is a test that is performed to make sure that previously working functionality still works, after changes elsewhere in the system. Your unit tests are automatically regression tests, and that's one of their biggest advantages. Once those tests are written, they will be run in future, whenever you add new functionality or change existing functionality. You don't need to explicitly write regression tests.

* The intent of regression testing is to provide a general assurance that no additional errors were introduced in the process of fixing other problems.
* -1 This definition is extremely broad. Every test makes sure working functionality still works - that's the general point of testing. Regression tests are tests written during debugging to make sure the code does not regress. “A test that was written when a bug was fixed. It ensure that this specific bug will not occur again. The full name is "non-regression test".

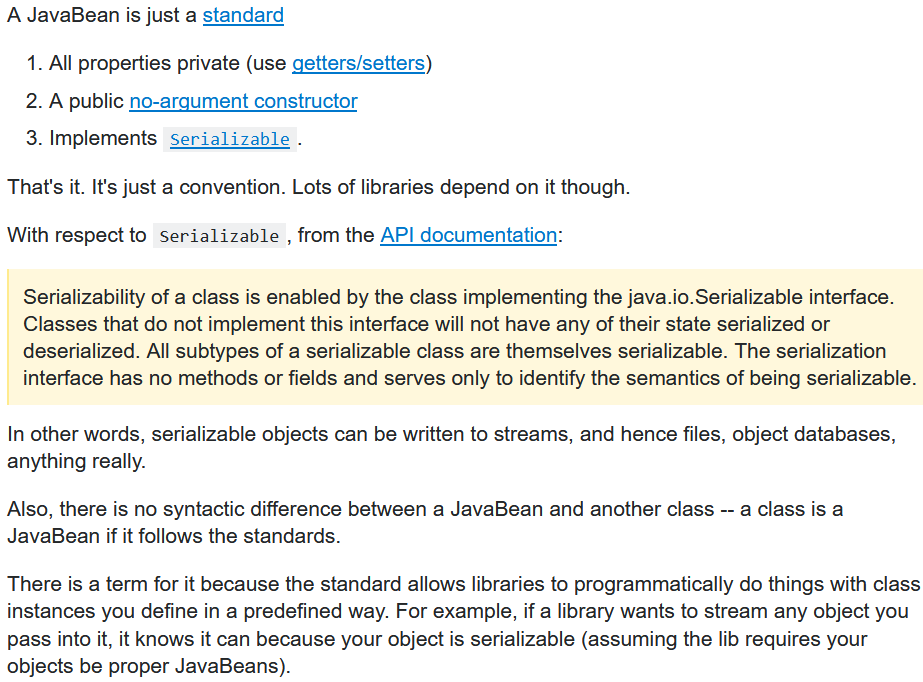
Notwithstanding the old joke, "Congress" is not the opposite of "progress;" "regress" is. For your code to regress is for it to "move backward," typically meaning that some bad behavior it once had, which you fixed, has come back. A "regression" is the return of a bug (although there can be other interpretations). A regression test, therefore, is a test that validates that you have fixed the bug, and one that you run periodically to ensure that your fix is still in place, still working.

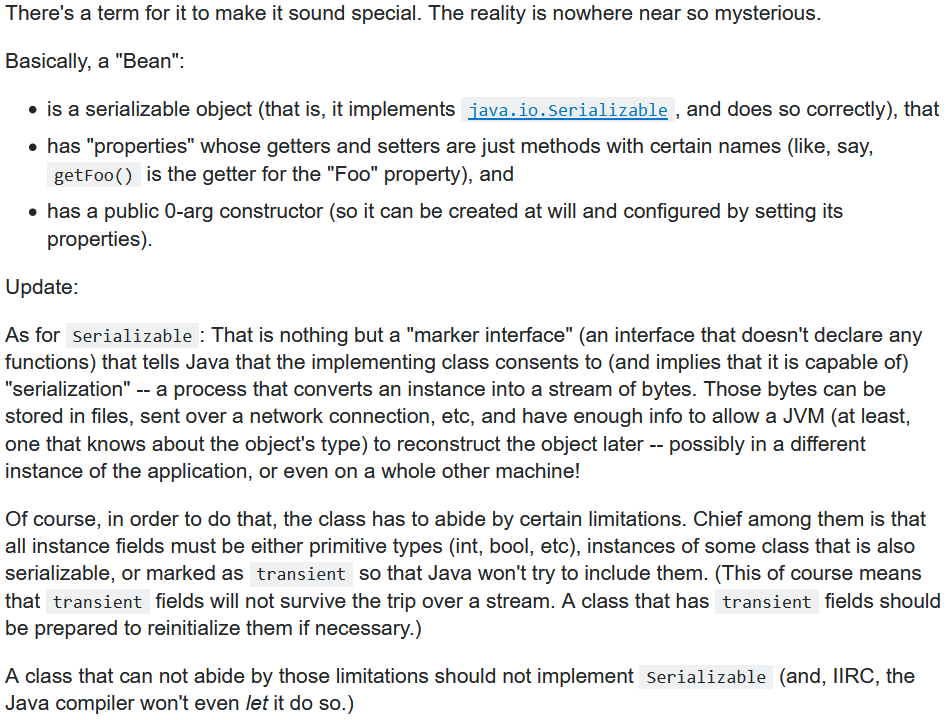
**Other Kinds of Tests**

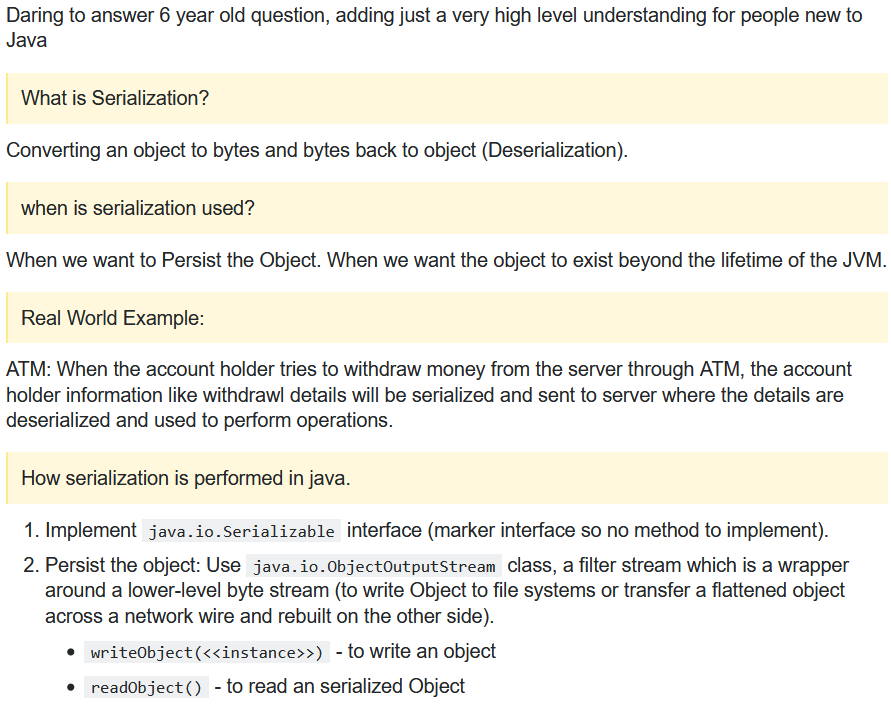


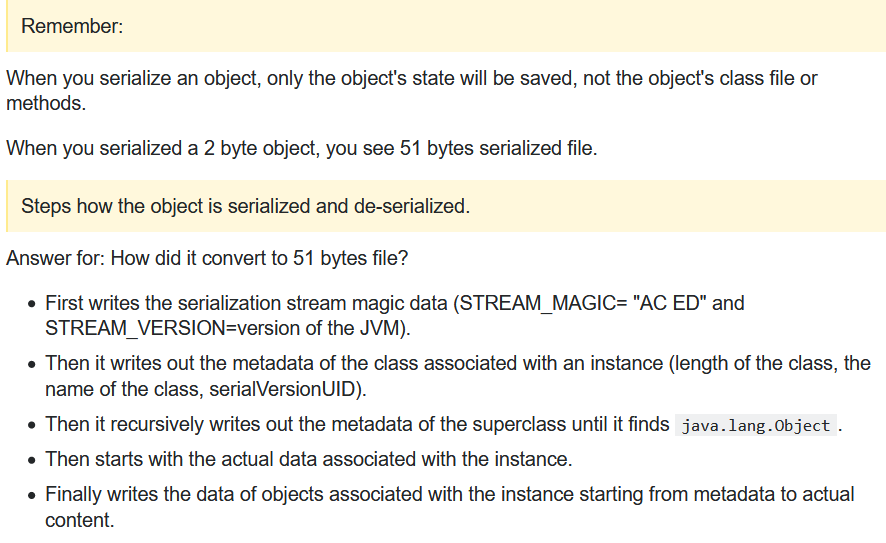


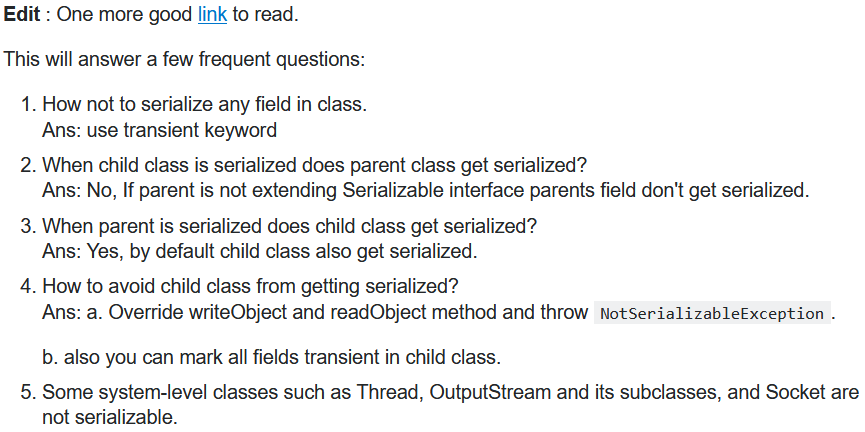
**What is a JavaBean exactly? / Java Serialization**





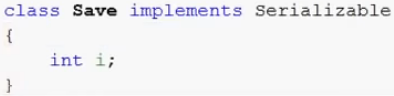


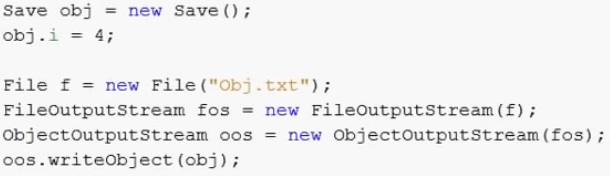




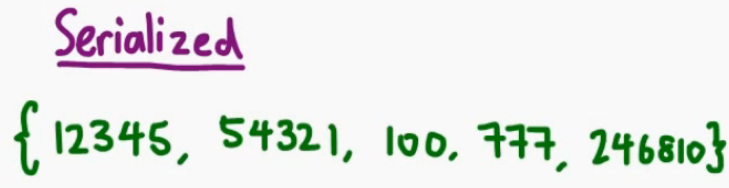
In computer science, in the context of data storage, serialization (or serialisation) is the process of translating data structures or object state into a format that can be stored (for example, in a file or memory buffer) or transmitted (for example, across a network connection link) and reconstructed later (possibly in a different computer environment). When the resulting series of bits is reread **according to the serialization format**, it can be used to create a semantically identical clone of the original object. For many complex objects, such as those that make extensive use of references, this process is not straightforward. Serialization of object-oriented objects **does not include** any of their associated **methods** with which they were previously linked.

This process of serializing an object is also called **marshalling** an object. The opposite operation, extracting a data structure from a series of bytes, is **deserialization** (also called **unmarshalling**).

**Simple Example**:





* There are many libraries in Java that support serialization of Java objects to JSON and back. json-lib, Json-io, Flexjson, Jackson, Gson ...

For simple data structures, for example an integer array might be serialized like this:

Whereas a more complicated structure with nested arrays, nested objects etc might be converted to this:

